Docket No.: 042390.P11280

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of: Guei-Yuan Lueh Application No.: 09/823,105 Filed: March 30, 2001 For: STATIC COMPILATION OF INSTRUMENTATION CODE FOR DEBUGGING SUPPORT	Examiner: Mary J. Steelman Art Group: 2122

APPEAL BRIEF

Mail Stop Appeal Brief-Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir:

Applicant submits the following Appeal Brief pursuant to 37 C.F.R. § 41.37 for consideration by the Board of Patent Appeals and Interferences. Applicant also submits herewith our check number 31562 in the amount of $500.00 to cover the cost of filing the opening brief as required by 37 C.F.R. §1.17(c). Please charge any additional fees or credit any overpayment to our deposit Account No. 02-2666. A duplicate copy of the Fee Transmittal is enclosed for this purpose.

TABLE OF CONTENTS

I. REAL PARTY IN INTEREST

The real party in interest is the assignee, Intel Corporation.

II. RELATED APPEALS AND INTERFERENCES

There are no related appeals or interferences known to the appellants, the appellants' legal representative, or assignee, which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

III. STATUS OF CLAIMS

Claims 1-38 of the present application are pending and remain rejected. The Applicant hereby appeals the rejection of claims 1-38.

IV. STATUS OF AMENDMENTS

The Applicant filed an amendment on September 29, 2004, in response to a Final Office Action issued by the Examiner on August 6, 2004. In response to the September 29, 2004 amendment, the Examiner issued an Advisory Action on November 5, 2004. The Applicant filed a Notice of Appeal from the Advisory Action issued by the Examiner on November 6, 2004.

V. SUMMARY OF CLAIMED SUBJECT MATTER

1. Independent claims 1, 16, and 31:

A debug support 370 includes a stack frame access support 410, a data value access support 420, a control break-point support 430, and a data break-point support 440[1]. The data break-point support 440 includes a field access and modifications watch 445 to provide support for field watches [2]. The field access and modification watch 445 has three models or approaches static, semi-static, and dynamic [3]. The static and semi-static data access support 445 includes a compilation 505 generation of an instrumentation code 510,

[1] Specification, paragraph [0031]; Figure 4.
[2] Specification, paragraph [0038]; Figure 4.
[3] Specification, paragraph [0044].

a guard of execution 530, and an insertion of instrumentation code 550[4]. The compilation 505 compiles a function that is in byte code format to produce native code. The compiled function occupies a code space[5]. Field access events are generated when the field specified is about to be accessed. Field accesses from Java Language code or from JNI are watched. Field accessed by other means is not watched[6]. The generation 510 generates an instrumentation code corresponding to a field watch of a field[7]. The guard 530 guards execution of the instrumentation code if the field watch is not activated[8]. The insertion 550 inserts the instrumentation code to the native code from the compilation 505[9].

2. Dependent claims 2, 17, and 32:

The generation 510 executes a field watch sequence[10]. For the static model, the execution of the field watch sequence is performed whether or not the field watch is activated. This field watch sequence includes the instruction sequence to spill the mimic stack operands, which are live at the field access point, to their canonical spill locations.

3. Dependent claims 3, 18, and 33:

For the static model, the guard 530 determines if a field watch is activated by comparing a flag with a predetermined watch value[11].

4. Dependent claims 4-5, 19-20, and 34-35:

For the static model, the instrumentation code is inserted before the field access or modification points. For the semi-static model, the instrumentation code is contained in a stub located at the end of the method's code space, after the original byte code sequence is compiled and emitted[12]. For the static model, the insertion 550 inserts the instrumentation code before a field access or modification point. For the semi-static model, the insertion 550 inserts the instrumentation code at the end of the code space of the compiled function[13].

5. Dependent claims 6-7, 21-22, and 36-37:

For the semi-static model, the guard 530 has two options: update the offset of a jump instruction to a stub having the field watch sequence, or replace a no-op (NOP)

---

[4] Specification, paragraph [0045]; Figure 5.
[5] Specification, paragraph [0046].
[6] Specification, paragraph [0039].
[7] Specification, paragraph [0047], first sentence.
[8] Specification, paragraph [0048], first sentence.
[9] Specification, paragraph [0049], first sentence.
[10] Specification, paragraph [0047]; Figure 5.
[11] Specification, paragraph [0048]; Figure 5, block 535.
[12] Specification, paragraph [0045]; Figure 5, blocks 552 and 554.

sequence with a jump instruction to the stub having the field watch sequence. Either option occurs at run time when the field watches are activated[14].

6. Dependent claims 8-11, 23-26, and 38:

The execution of the field watch sequence 520 includes saving live global state executing an event hook function for an event corresponding to the field watch, and restoring the live global state. The live global state corresponds to an active register such as scratch registers or floating-point registers[15].

7. Dependent claims 12-15, and 27-30:

In one embodiment, the function is a Java method[16], the field is a Java field and the field watch is a Java field watch as specified in the JVMDI[17]. The guard operation takes advantage of the flag usage in the JVM. The JVM stores a Boolean field for each class field in its internal data structures. This Boolean flag is set to true when the field watch is activated and to false when the field watch is cleared[18].

VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

1. Independent claims 1, 16, and 31, and dependent claims 4-5, 13-15, 19-20, and 28-30, 34-35 stand rejected under 35 U.S.C. §103(a) as being unpatentable over U.S. Pre Grant Publication No. 2001/0047510A1 issued to Angel et al. ("Angel") in view of "Poor Man's Watchpoints", by Max Copperman and Jeff Thomas (1995) ("Copperman").
2. Dependent claims 2, 17, and 32 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.
3. Dependent claims 3, 18, and 33 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.
4. Dependent claims 6, 21, and 36 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.
5. Dependent claims 7, 22, and 37 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.

---

[13] Specification, paragraph [0049]; Figure 5, blocks 552 and 554.
[14] Specification, paragraph [0048]; Figure 5, blocks 542 and 544.
[15] Specification, paragraph [0047]; Figure 5, blocks 522, 524, and 526.
[16] Specification, paragraph [0046].
[17] Specification, paragraph [0047].
[18] Specification, paragraph [0062].

6. Dependent claims 8, 23, and 38 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.
7. Dependent claims 9, 11, 24, and 26 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.
8. Dependent claims 10 and 25 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.
9. Dependent claims 12 and 27 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Angel in view of Copperman.

VII. ARGUMENTS

1. Independent claims 1, 16, and 31, and dependent claims 4-5, 13-15, 19-20, and 28-30, 34-35 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner rejected claims 1-38 under 35 U.S.C. §103(a) as being unpatentable over U.S. Pre Grant Publication No. 2001/0047510A1 issued to Angel et al. ("Angel") in view of "Poor Man's Watchpoints", by Max Copperman and Jeff Thomas (1995) ("Copperman"). Applicant respectfully traverses the rejection and contends that the Examiner has not met the burden of establishing a prima facie case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. *MPEP §2143, p. 2100-129 (8th Ed., Rev. 2, May 2004).* Applicant respectfully contends that there is no suggestion or motivation to combine their teachings, and thus no *prima facie* case of obviousness has been established.

Angel discloses a byte code instrumentation. A technique to instrument a byte code program includes examining the byte code, selecting portions of the byte code for instrumentation, and instrumenting the portions to provide instrumented byte code (Angel, paragraph [0014]). Memory access instructions are instrumented to detect illegal memory operations at runtime (Angel, paragraph [0091]. In addition, exiting and entering blocks of

code where variables become defined and undefined are monitored (Angel, paragraph [0091]). Angel does not disclose a field watch.

Copperman discloses a technique to implement watch points using code patching. When the user sets a watch point, the debugger sets the register $fp to point to a register save area in the debuggee's static data space. When no watch points are set, the first instruction in the patch branches around the rest of the patch if $fp contains (Copperman, page 38, third paragraph under section "The Debuggee"). Watchpoints in Copperman allow the user to get control at the point that the location is written to, that is, at the assignment through the pointer, allowing the user to identify the errant pointer (Copperman, first paragraph under the section "Introduction"). Since Copperman's watchpoint refers to a memory location referenced by a pointer, it is not the same as a field watch.

Angel and Copperman, taken alone or in any combination, does not disclose, suggest, or render obvious (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. There is no motivation to combine Angel and Copperman because neither of them addresses the problem of compilation according to a field watch. There is no teaching or suggestion that guarding execution of instrumentation code or a field byte code accessing or modifying the field is present. Angel, read as a whole, does not suggest the desirability of generating an instrumentation code corresponding to the field watch.

The Examiner states that Angel discloses, generating an instrumentation code corresponding to a field watch of the field (Final Office Action, page 3). Applicant respectfully disagrees. The cited paragraph, [0125], merely states that "[d]escribed below are methods of automatically editing the executable byte code representation...for generating instrumented byte code" (Angel, page 11, paragraph [0125]). There is no teaching or suggestion on the use of a field watch in generating instrumentation code. A field watch sequence may include instruction sequence to spill the mimic stack operands, which are live at the field access point, to their canonical spill locations (See, for example, Specification, page 14, paragraph [0047]).

The Examiner further states that Copperman discloses guarding execution of the instrumentation code if the field watch is not activated by disclosing setting or not setting the watchpoints, or entering or enabling a watchpoint command (Final Office Action, page 4). Applicant respectfully disagrees. A watchpoint or watchpoint command is not the same as a field watch as discussed above.

In the final Office Action, the Examiner responds to the Applicant's arguments (final Office Action, pages 10-14). Applicant contends that the Examiner's responses failed to overcome Applicant's arguments.

The Examiner states that Angel and Copperman are justifiably combined as both and related to instrumentation of code in order to facilitate debugging (Final Office Action, page 11, item (A)). However, neither Angel nor Copperman discloses or suggests a field byte code that accesses or modifies a field and generating an instrumentation code corresponding to a field watch of the field. Therefore, the combination of Angel nor Copperman is improper.

The Examiner further states that compilation according to a field watch is not a claim limitation (Final Office Action, page 12, item (C)). Applicant respectfully disagrees. Applicant contends that neither Angel nor Copperman addresses the problem of compilation according to a field watch in that there is an operation of generating an instrumentation code corresponding to a field watch of the field.

The Examiner further states that an instruction being instrumented could be a field watch, i.e., related to memory variable access (Final Office Action, page 13, item (D)). Applicant respectfully disagrees. A memory variable access is not a field watch. A field is a Java variable that is defined in a Java object. Field access events are generated when the field specified is about to be accessed. Field accesses from Java Language Code or from JNI are watched (See Specification, page 12, paragraph [0039].

The Examiner further states that a Web definition states a watchpoint is a type of breakpoint that is triggered whenever the class field being monitored is modified. However, the Examiner did not produce evidence of this definition. Furthermore, even if that definition is correct, it does not mean that Copperman discloses or suggests the watchpoint in that context. Copperman discloses that watchpoints aid in finding bugs that caused by an assignment through an unidentified pointer modifying an arbitrary memory

location (Copperman, page 37, first paragraph under section heading "Introduction"). Therefore, a watchpoint in Copperman is not the same as the field watch.

The Examiner further offers the definition from "JAVA Debug Interface". However, there is no linkage between this definition and Copperman.

In the Advisory Action dated November 5, 2004, the Examiner, citing paragraphs [0125] and [0127] in Angel, states that Angel provided references that determine if a field is to be "watched", then instrumentation code is generated and inserted. Applicant respectfully disagrees for the following reasons.

For ease of reference, the cited paragraphs in Angel are copied below.

> *[0125] Other embodiments also exist. Described below are methods of automatically editing the executable byte code representation of a computer program or other methods for generating instrumented byte code. In one embodiment, the byte code is altered by the addition of new instructions and/or the deletion or modification of existing instructions.*
>
> *[0127] One objective of the instrumentation process is to alter the program to facilitate the gathering of diagnostic and statistical information on the program when it is executed; i.e., dynamic analysis. This allows the program's internal state to be monitored for variety of purposes. These purposes include, but are not limited to: diagnosing error conditions that occur at run time, creating a record of the inner details of program execution, measuring program execution to provide code coverage analysis and performance profiling, or providing additional run time error or exception handling.*

As shown in the above, there is no reference of a field watch. The reference that "[t]his allows the program's internal state to be monitored for variety of purposes" in paragraph [0127] merely refers to the program's internal state, not a field watch. As discussed above, a memory variable access is not a field watch. A field is a Java variable that is defined in a Java object. Field access events are generated when the field specified

is about to be accessed. Field accesses from Java Language Code or from JNI are watched (See Specification, page 12, paragraph [0039].

2. Dependent claims 2, 17, and 32 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Copperman discloses executing a field watch sequence, citing "[o]n receiving a watchpoint command, the debugger has to add an entry to the watch table and ensure that <cmd> is executed when the watchpoint is hit" (Copperman, page 40, "The Debugger"). Applicant respectfully disagrees. Copperman merely discloses <cmd> is executed when the watchpoint is hit. <cmd> is merely a watchpoint command, not "a field watch sequence" as part of "generating an instrumentation code corresponding to a field watch of the field". Furthermore, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious generating the instrumentation code comprising executing a field watch sequence.

3. Dependent claims 3, 18, and 33 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Copperman discloses comparing a flag with a predetermined watch value to determine if the field watch is activated, citing "[a] flag passed to the post-loader designates loads, stores, or both as patch targets" (Copperman, bottom of page 38). Applicant respectfully disagrees. Copperman's flag here is used merely to designate loads, stores, or both as patch targets, not to determine if the field watch is activated.

Furthermore, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native

code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious guarding execution of the instrumentation code comprising comparing a flag with a predetermined watch value to determine if the field watch is activated.

4. Dependent claims 6, 21, and 36 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Angel discloses updating an offset of a jump instruction to a stub having the field watch sequence when the field watch is activated, citing paragraphs [0176], [0181], [0182], [0119], and [0091]. However, none of these paragraphs discloses: (1) a stub having the field watch sequence, (2) updating an offset of a jump instruction to the stub, and (3) when the field watch is activated. Angel merely discloses the addresses of the functions are set to stub routines that simply return without executing any thing (Angel, paragraph [0119]). Since Angel's stub routines simply return without any execution, it cannot have the field watch sequence. Furthermore, Angel merely discloses: (1) the code table being indexed by an offset or the code table being modified to reflect the new offset (Angel, paragraphs [0176] and [0181]), not an offset of a jump instruction; (2) the byte code being modified to update branch instructions to reflect the new offsets, not an offset of a jump instruction. In addition, these offsets or branch instructions are not related to the stub having the field watch sequence.

Finally, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious guarding execution of the instrumentation code comprising updating an offset of a jump instruction to a stub having the field watch sequence when the field watch is activated.

5. Dependent claims 7, 22, and 37 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Copperman discloses replacing a no-op sequence with a jump instruction to a stub having

the field watch sequence when the field watch is activated, citing page 37, 4th paragraph of section "Introduction"; page 40, last paragraph, and page 41, third paragraph. Applicant respectfully disagrees. Copperman merely discloses adding code to modify the parser, to disable, enable, and cancel individual watchpoints (Copperman, page 41, third paragraph), not a no-op sequence. Furthermore, none of the cited paragraphs discloses: (1) a stub having the field watch sequence, (2) replacing a no-op sequence with a jump instruction to the stub, and (3) when the field watch is activated.

Finally, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious guarding execution of the instrumentation code comprising replacing a no-op sequence with a jump instruction to a stub having the field watch sequence when the field watch is activated.

6. Dependent claims 8, 23, and 38 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Angel discloses: (1) saving live global state, the live global state corresponding to an active register, and (2) executing an event hook function for an event corresponding to the field watch; and (3) restoring the live global state, citing paragraphs [0169], [0170], [0174], and [0137]. Applicant respectfully disagrees. Angel merely discloses the patch using an assembly code thunk that includes a small amount of assembly code and a class instance that lets the patch code control before the native code routine starts. This is not the same as "saving". Furthermore, getting control is not the same as saving live global state, or restoring the live global state. In addition, "taking advantage of particular hooks" is not the same as executing an event hook function.

Finally, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if

the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious executing the field watch sequence comprising (1) saving live global state, the live global state corresponding to an active register, and (2) executing an event hook function for an event corresponding to the field watch; and (3) restoring the live global state.

7. Dependent claims 9, 11 and 24, and 26 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Angel discloses pushing the live global state onto a stack, and retrieving the live global state from the stack, citing paragraph [0153]. Applicant respectfully disagrees. Passing parameters using a stack is not the same as pushing the live global state. Parameters are not the live global state.

Furthermore, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious saving the live global state comprising pushing the live global state onto a stack, and retrieving the live global state from the stack.

8. Dependent claims 10 and 25 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Angel discloses (1) passing an argument corresponding to the field, and (2) calling a run-time library related to the event, citing paragraphs [0093], [0114], [0113], and [0119]. Applicant respectfully disagrees. None of the cited paragraphs discloses executing the event hook function. The child node as one of the arguments is not an argument corresponding to the field. The run-time instrumentation code is not related to an event corresponding to the field watch.

Furthermore, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1)

compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious executing the event hook function comprising (1) passing an argument corresponding to the field, and (2) calling a run-time library related to the event.

9. Dependent claims 12 and 27 Are Not Obvious Over Angel In View of Copperman:

In the final Office Action dated August 6, 2004, the Examiner states that Copperman discloses activating/ clearing the field watch by setting/ resetting the flag, citing page 40, second paragraph of the section "Maintaining The Watch Table", and page 41, third paragraph. Applicant respectfully disagrees. Disabling or clearing a command is not the same as activating or clearing a field watch. A command is not a field watch. Furthermore, as argued above, a watchpoint is not the same as a field watch.

Finally, as argued above in the independent claims 1, 16, and 31, Angel and Copperman, taken alone or in combination, do not disclose or render obvious: (1) compiling a function including a byte code sequence, (2) generating an instrumentation code corresponding to a field watch, (3) guarding execution of the instrumentation code if the field watch is not activated; and (4) inserting the instrumentation code to the native code. Accordingly, Angel and Copperman, taken alone or in combination, do not disclose or render obvious activating/ clearing the field watch by setting/ resetting the flag.

The Examiner failed to establish a prima facie case of obviousness and failed to show there is teaching, suggestion or motivation to combine the references. "When determining the patentability of a claimed invention which combined two known elements, 'the question is whether there is something in the prior art as a whole suggest the desirability, and thus the obviousness, of making the combination.'" In re Beattie, Lindemann Maschinenfabrik GmbH v. American Hoist & Derrick Co., 730 F.2d 1452, 1462, 221 USPQ (BNA) 481, 488 (Fed. Cir. 1984). To defeat patentability based on obviousness, the suggestion to make the new product having the claimed characteristics must come from the prior art, not from the hindsight knowledge of the invention. Interconnect Planning Corp. v. Feil, 744 F.2d 1132, 1143, 227 USPQ (BNA) 543, 551 (Fed. Cir. 1985). To prevent the use of hindsight based on the invention to defeat

patentability of the invention, this court requires the Examiner to show a motivation to combine the references that create the case of obviousness. In other words, the Examiner must show reasons that a skilled artisan, confronted with the same problems as the inventor and with no knowledge of the claimed invention, would select the prior elements from the cited prior references for combination in the manner claimed. In re Rouffet, 149 F.3d 1350 (Fed. Cir. 1996), 47 USPQ 2d (BNA) 1453. "To support the conclusion that the claimed invention is directed to obvious subject matter, either the references must expressly or implicitly suggest the claimed invention or the Examiner must present a convincing line of reasoning as to why the artisan would have found the claimed invention to have been obvious in light of the teachings of the references." Ex parte Clapp, 227 USPQ 972, 973. (Bd.Pat.App.&Inter. 1985). The mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination. In re Mills, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990). Furthermore, although a prior art device "may be capable of being modified to run the way the apparatus is claimed, there must be a suggestion or motivation in the reference to do so." In re Mills 916 F.2d at 682, 16 USPQ2d at 1432; In re Fitch, 972 F.2d 1260, 23 USPQ2d 1780 (Fed. Cir. 1992).

In the present invention, the cited references do not expressly or implicitly suggest any one of the above elements. In addition, the Examiner failed to present a convincing line of reasoning as to why a combination of Angel and Copperman is an obvious application of using the field watch in debugger support.

Therefore, Applicant believes that independent claims 1, 16, 31 and their respective dependent claims are distinguishable over the cited prior art references.

VIII. CONCLUSION

Applicant respectfully requests that the Board enter a decision overturning the Examiner's rejection of all pending claims, and holding that the claims are neither anticipated nor rendered obvious by the prior art.

Respectfully submitted,

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP

Dated: January 11, 2005

THINH V. NGUYEN
Reg. No. 42,034

12400 Wilshire Blvd., 7th Floor
Los Angeles, CA 90025-1026
(714) 557-3800

IX. CLAIMS APPENDIX

The claims of the present application which are involved in this appeal are as follows:

1. (original) A method comprising:

compiling a function including a byte code sequence having a field byte code that accesses or modifies a field, the compiled function providing a native code and occupying a code space;

generating an instrumentation code corresponding to a field watch of the field;

guarding execution of the instrumentation code if the field watch is not activated; and

inserting the instrumentation code to the native code.

2. (original) The method of claim 1 wherein generating the instrumentation code comprises:

executing a field watch sequence.

3. (original) The method of claim 2 wherein guarding execution of the instrumentation code comprises:

comparing a flag with a predetermined watch value to determine if the field watch is activated.

4. (original) The method of claim 3 wherein inserting the instrumentation code comprises:

inserting the instrumentation code before a field access or modification point.

5. (original) The method of claim 2 wherein inserting the instrumentation code comprises:

inserting the instrumentation code at end of the code space.

6. (original) The method of claim 5 wherein guarding execution of the instrumentation code comprises:

updating an offset of a jump instruction to a stub having the field watch sequence when the field watch is activated.

7. (original) The method of claim 5 wherein guarding execution of the instrumentation code comprises:

replacing a no-op sequence with a jump instruction to a stub having the field watch sequence when the field watch is activated.

8. (original) The method of claim 2 wherein executing the field watch sequence comprises:

saving live global state, the live global state corresponding to an active register;

executing an event hook function for an event corresponding to the field watch; and

restoring the live global state.

9. (original) The method of claim 8 wherein saving the live global state comprises:

pushing the live global state onto a stack.

10. (original) The method of claim 8 wherein executing the event hook function comprises:

passing an argument corresponding to the field; and

calling a run-time library function related to the event.

11. (original) The method of claim 9 wherein restoring the live global state comprises:

retrieve the live global state from the stack.

12. (original) The method of claim 3 further comprising:

activating the field watch by setting the flag; and

clearing the field watch by resetting the flag.

13. (previously presented) The method of claim 1 wherein the function is a JAVA method.

14. (previously presented) The method of claim 1 wherein the field is a JAVA field in a JAVA virtual machine.

15. (previously presented) The method of claim 8 wherein the event hook function is compatible with a JAVA Virtual Machine Debug Interface (JVMDI).

16. (original) A computer program product comprising:

a machine useable medium having computer program code embedded therein, the computer program product having:

computer readable program code to compile a function including a byte code sequence having a field byte code that accesses or modifies a field, the compiled function providing a native code occupying a code space;

computer readable program code to generate an instrumentation code corresponding to a field watch of the field;

computer readable program code to guard execution of the instrumentation code if the field watch is not activated; and

computer readable program code to insert the instrumentation code to the native code.

17. (original) The computer program product of claim 16 wherein the computer readable program code to generate the instrumentation code comprises:

computer readable program code to execute a field watch sequence.

18. (original) The computer program product of claim 17 wherein the computer readable program code to guard execution of the instrumentation code comprises:

computer readable program code to compare a flag with a predetermined watch value to determine if the field watch is activated.

19. (original) The computer program product of claim 18 wherein the computer readable program code to insert the instrumentation code comprises:

computer readable program code to insert the instrumentation code before a field access or modification point.

20. (original) The computer program product of claim 17 wherein the computer readable program code to insert the instrumentation code comprises:

computer readable program code to insert the instrumentation code at end of the code space.

21. (original) The computer program product of claim 20 wherein the computer readable program code to guard execution of the instrumentation code comprises:

computer readable program code to update an offset of a jump instruction to a stub having the field watch sequence when the field watch is activated.

22. (original) The computer program product of claim 20 wherein the computer readable program code to guard execution of the instrumentation code comprises:

computer readable program code to replace a no-op sequence with a jump instruction to a stub having the field watch sequence when the field watch is activated.

23. (original) The computer program product of claim 17 wherein the computer readable program code to execute the field watch sequence comprises:

computer readable program code to save live global state, the live global state corresponding to an active register;

computer readable program code to execute an event hook function for an event corresponding to the field watch; and

computer readable program code to restore the live global state.

24. (original) The computer program product of claim 23 wherein the computer readable program code to save the live global state comprises:

computer readable program code to push the live global state onto a stack.

25. (original) The computer program product of claim 23 wherein the computer readable program code to execute the event hook function comprises:

computer readable program code to pass an argument corresponding to the field; and

computer readable program code to call a run-time library function related to the event.

26. (original) The computer program product of claim 24 wherein the computer readable program code to restore the live global state comprises:

computer readable program code to retrieve the live global state from the stack.

27. (original) The computer program product of claim 18 further comprising:

computer readable program code to activate the field watch by setting the flag; and

computer readable program code to clear the field watch by resetting the flag.

28. (previously presented) The computer program product of claim 16 wherein the function is a JAVA method.

29. (previously presented) The computer program product of claim 16 wherein the field is a JAVA field in a JAVA virtual machine.

30. (previously presented) The computer program product of claim 23 wherein the event hook function is compatible with a JAVA Virtual Machine Debug Interface (JVMDI).

31. (original) A system comprising:

a processor;

a memory coupled to the processor, the memory storing instruction code, the instruction code, when executed by the processor, causing the processor to:

compile a function including a byte code sequence having a field byte code that accesses or modifies a field, the compiled function providing a native code and occupying a code space,

generate an instrumentation code corresponding to a field watch of a field,

guard execution of the instrumentation code if the field watch is not activated, and

insert the instrumentation code to the native code.

32. (original) The system of claim 31 wherein the instruction code causing the processor to generate the instrumentation code causes the processor to:

execute a field watch sequence.

33. (original) The system of claim 32 wherein the instruction code causing the processor to guard execution of the instrumentation code causes the processor to:

compare a flag with a predetermined watch value to determine if the field watch is activated.

34. (original) The system of claim 33 wherein the instruction code causing the processor to insert the instrumentation code causes the processor to:

insert the instrumentation code before a field access or modification point.

35. (original) The system of claim 32 wherein the instruction code causing the processor to insert the instrumentation code causes the processor to:

insert the instrumentation code at end of the code space.

36. (original) The system of claim 35 wherein the instruction code causing the processor to guard execution of the instrumentation code causes the processor to:

update an offset of a jump instruction to a stub having the field watch sequence when the field watch is activated.

37. (original) The system of claim 35 wherein the instruction code causing the processor to guard execution of the instrumentation code causes the processor to:

replace a no-op sequence with a jump instruction to a stub having the field watch sequence when the field watch is activated.

38. (original) The system of claim 32 wherein the instruction code causing the processor to execute the field watch sequence causes the processor to:

save live global state, the live global state corresponding to an active register;

execute an event hook function for an event corresponding to the field watch; and

restore the live global state.